



# Autoencoders and Restricted Boltzmann Machines

Amir H. Payberah  
[payberah@kth.se](mailto:payberah@kth.se)  
27/11/2019



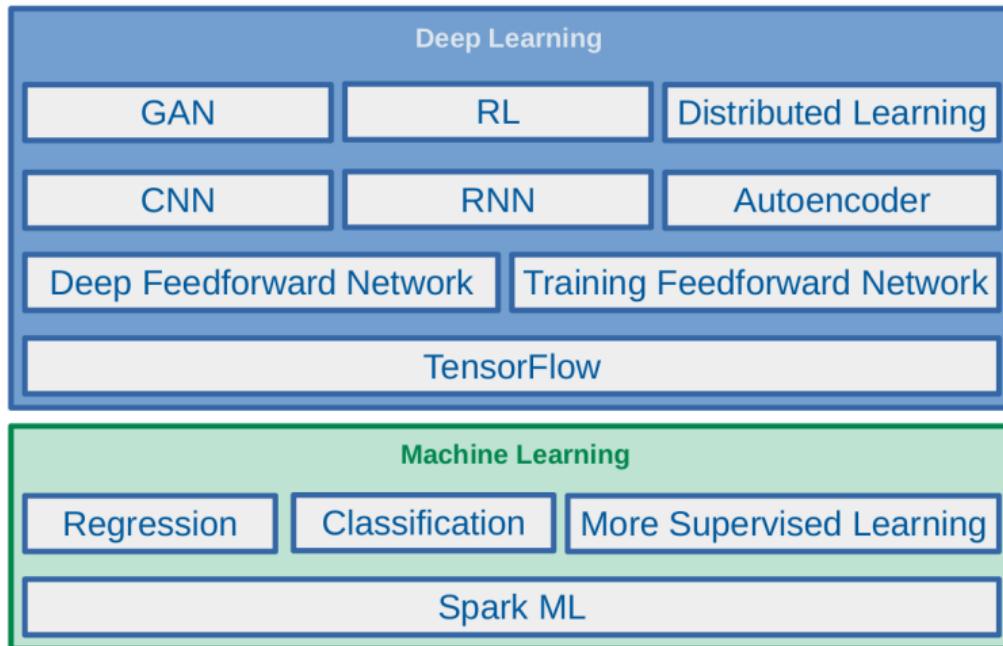


# The Course Web Page

<https://id2223kth.github.io>

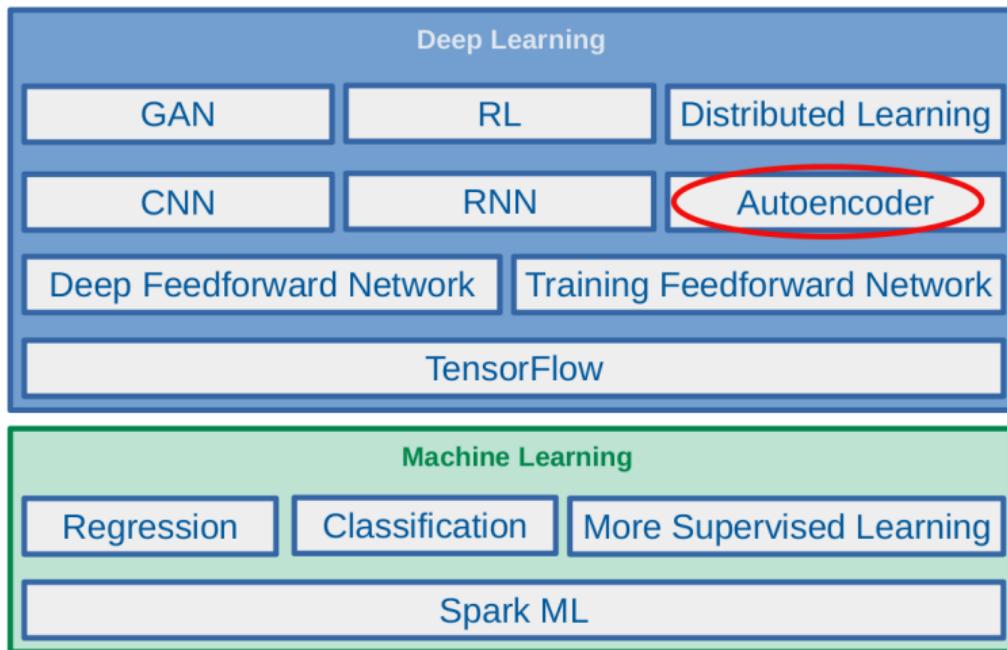


# Where Are We?





# Where Are We?





# Let's Start With An Example

- ▶ Which of them is easier to memorize?
- ▶ Seq1: 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
- ▶ Seq2: 50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20

Seq1 : 40, 27, 25, 36, 81, 57, 10, 73, 19, 68

Seq2 : 50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20

- ▶ Seq1 is shorter, so it should be easier.
- ▶ But, Seq2 follows two simple rules:
  - Even numbers are followed by their half.
  - Odd numbers are followed by their triple plus one.
- ▶ You don't need pattern if you could quickly and easily memorize very long sequences
- ▶ But, it is hard to memorize long sequences that makes it useful to recognize patterns.



- ▶ 1970, W. Chase and H. Simon
- ▶ They observed that **expert chess players** were able to **memorize** the positions of **all** the pieces in a game by looking at the board for just **5 seconds**.



- ▶ This was only the case when the pieces were placed in **realistic positions**, not when the pieces were placed **randomly**.
- ▶ Chess experts **don't have a much better memory** than you and I.
- ▶ They just see chess **patterns more easily** due to their **experience** with the game.
- ▶ **Patterns** helps them store information **efficiently**.

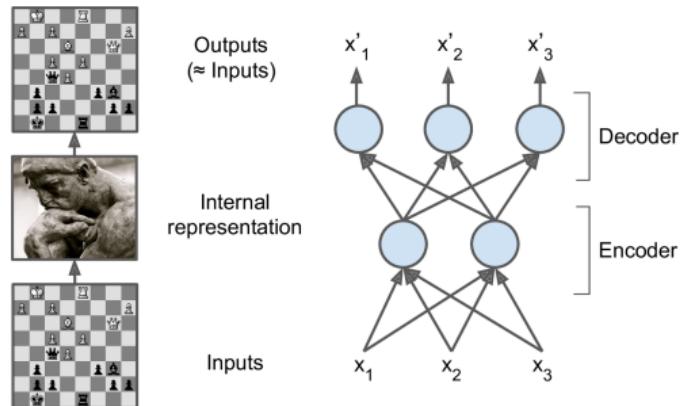




# Autoencoders

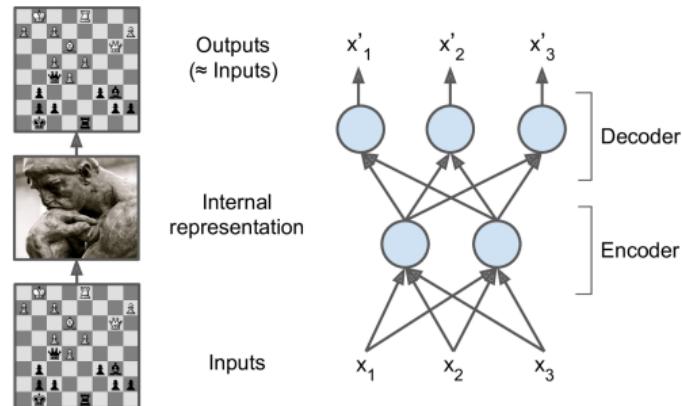
## Autoencoders (1/5)

- ▶ Just like the chess players in this memory experiment.
- ▶ An **autoencoder** looks at the inputs, **converts** them to an efficient **internal representation**, and then **spits out** something that **looks very close to the inputs**.



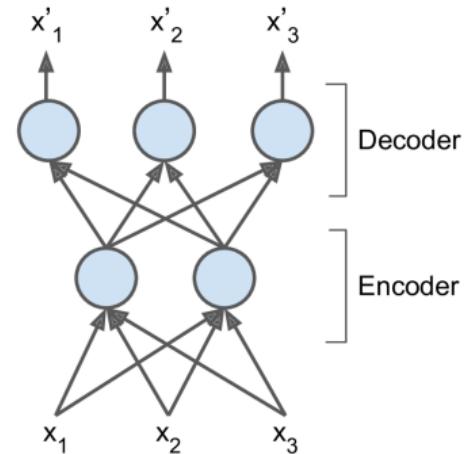
## Autoencoders (2/5)

- ▶ The same architecture as a Multi-Layer Perceptron (MLP).
- ▶ Except that the number of neurons in the output layer must be **equal** to the **number of inputs**.



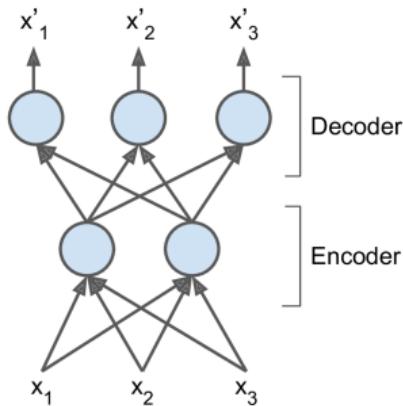
## Autoencoders (3/5)

- ▶ An autoencoder is always composed of **two parts**.
- ▶ An **encoder (recognition network)**,  $\mathbf{h} = f(\mathbf{x})$   
Converts the **inputs** to an internal representation.
- ▶ A **decoder (generative network)**,  $\mathbf{r} = g(\mathbf{h})$   
Converts the **internal representation** to the **outputs**.
- ▶ If an autoencoder learns to set  $g(f(\mathbf{x})) = \mathbf{x}$  everywhere, it is **not especially useful**, **why?**



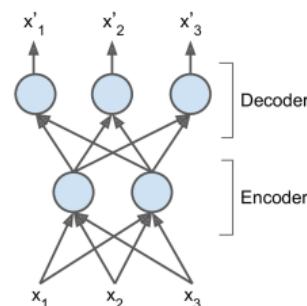
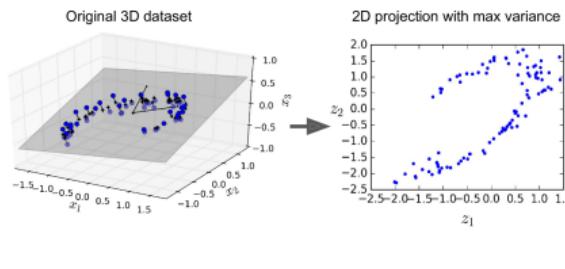
## Autoencoders (4/5)

- ▶ Autoencoders are designed to be **unable** to learn to copy perfectly.
- ▶ The models are forced to **prioritize which aspects of the input** should be copied, they often learn **useful properties** of the data.



## Autoencoders (5/5)

- ▶ Autoencoders are neural networks capable of learning efficient representations of the input data (called codings) without any supervision.
- ▶ Dimension reduction: these codings typically have a much lower dimensionality than the input data.





# Different Types of Autoencoders

- ▶ Stacked autoencoders
- ▶ Denoising autoencoders
- ▶ Sparse autoencoders
- ▶ Variational autoencoders

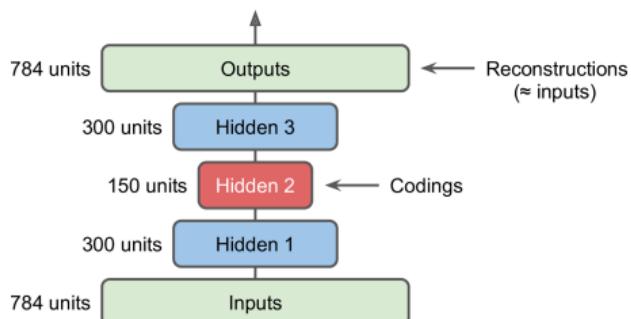


# Different Types of Autoencoders

- ▶ Stacked autoencoders
- ▶ Denoising autoencoders
- ▶ Sparse autoencoders
- ▶ Variational autoencoders

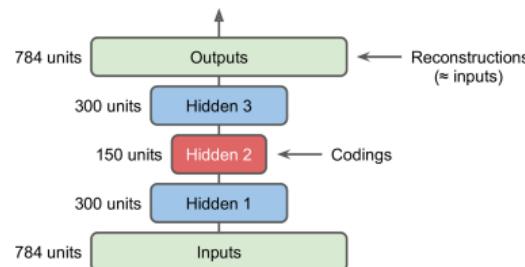
## Stacked Autoencoders (1/3)

- ▶ **Stacked autoencoder**: autoencoders with **multiple hidden layers**.
- ▶ Adding **more layers** helps the autoencoder learn more **complex codings**.
- ▶ The architecture is typically **symmetrical** with regards to the **central hidden layer**.



## Stacked Autoencoders (2/3)

- ▶ In a symmetric architecture, we can **tie the weights** of the **decoder layers** to the weights of the **encoder layers**.
- ▶ In a network with **N** layers, the **decoder layer weights** can be defined as  $w_{N-1+1} = w_1^T$ , with  $l = 1, 2, \dots, \frac{N}{2}$ .
- ▶ This **halves** the **number of weights** in the model, **speeding up training** and **limiting the risk of overfitting**.





## Stacked Autoencoders (3/3)

```
stacked_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(30, activation="relu"),
])
stacked_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="relu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
model = keras.models.Sequential([stacked_encoder, stacked_decoder])
```

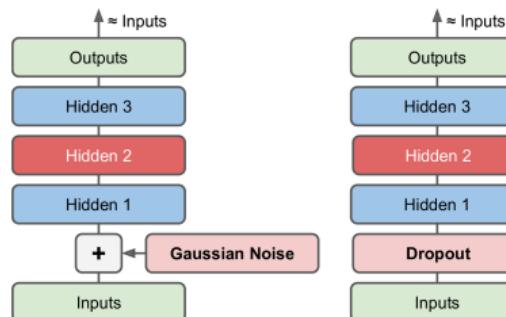


# Different Types of Autoencoders

- ▶ Stacked autoencoders
- ▶ Denoising autoencoders
- ▶ Sparse autoencoders
- ▶ Variational autoencoders

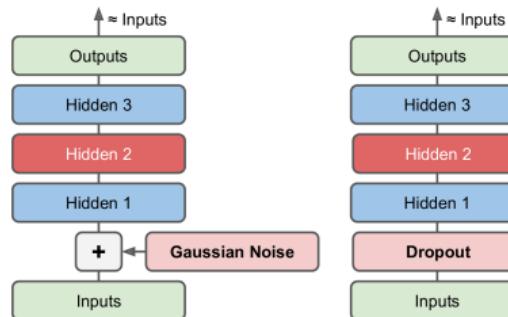
# Denoising Autoencoders (1/4)

- ▶ One way to force the autoencoder to **learn useful features** is to **add noise** to its **inputs**, training it to **recover the original noise-free inputs**.
- ▶ This prevents the autoencoder from **trivially copying** its **inputs** to its **outputs**, so it ends up having to find patterns in the data.



## Denoising Autoencoders (2/4)

- The noise can be pure Gaussian noise added to the inputs, or it can be randomly switched off inputs, just like in dropout.





## Denoising Autoencoders (3/4)

```
denoising_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(30, activation="relu")
])
denoising_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="relu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
model = keras.models.Sequential([denoising_encoder, denoising_decoder])
```



## Denoising Autoencoders (4/4)

```
denoising_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.GaussianNoise(0.2),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(30, activation="relu")
])
denoising_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="relu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
model = keras.models.Sequential([denoising_encoder, denoising_decoder])
```



# Different Types of Autoencoders

- ▶ Stacked autoencoders
- ▶ Denoising autoencoders
- ▶ **Sparse autoencoders**
- ▶ Variational autoencoders



## Sparse Autoencoders (1/2)

- ▶ Adding an appropriate term to the **cost function** to push the autoencoder to **reducing** the number of **active neurons** in the **coding layer**.
- ▶ This forces the autoencoder to represent each input as a combination of a **small number of activations**.
- ▶ As a result, **each neuron** in the **coding layer** typically ends up representing a **useful feature**.



## Sparse Autoencoders (2/2)

```
sparse_l1_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(300, activation="sigmoid", activity_regularizer=keras.regularizers.l1(1e-3))
])

sparse_l1_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[300]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

model = keras.models.Sequential([sparse_l1_encoder, sparse_l1_decoder])
```



# Different Types of Autoencoders

- ▶ Stacked autoencoders
- ▶ Denoising autoencoders
- ▶ Sparse autoencoders
- ▶ **Variational autoencoders**

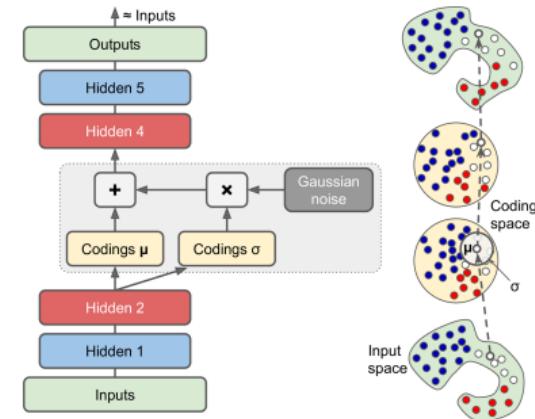


## Variational Autoencoders (1/6)

- ▶ Variational autoencoders are probabilistic autoencoders.
- ▶ Their outputs are partly determined by chance, even after training.
  - As opposed to denoising autoencoders, which use randomness only during training.
- ▶ They are generative autoencoders, meaning that they can generate new instances that look like they were sampled from the training set.

## Variational Autoencoders (2/6)

- ▶ Instead of directly producing a coding for a given input, the **encoder** produces a **mean coding  $\mu$**  and a **standard deviation  $\sigma$** .
- ▶ The **actual coding** is then **sampled randomly** from a **Gaussian distribution** with mean  $\mu$  and **standard deviation  $\sigma$** .
- ▶ After that the **decoder** just **decodes the sampled coding normally**.





## Variational Autoencoders (3/6)

- ▶ The **cost function** is composed of **two parts**.
- ▶ 1. the usual **reconstruction loss**.
  - Pushes the autoencoder to **reproduce its inputs**.
  - Using **cross-entropy**.
- ▶ 2. the **latent loss**
  - Pushes the autoencoder to have **codings** that look as though they were sampled from a **simple Gaussian distribution**.
  - Using the **KL divergence** between the **target distribution** (the Gaussian distribution) and the **actual distribution** of the codings.
  - $$\text{latent\_loss} = -\frac{1}{2} \sum_1^K (1 + \log(\sigma_i^2) - \sigma_i^2 - \mu_i^2)$$



## Variational Autoencoders (4/6)

### ► Encoder part

```
inputs = keras.layers.Input(shape=[28, 28])
z = keras.layers.Flatten()(inputs)
z = keras.layers.Dense(150, activation="relu")(z)
z = keras.layers.Dense(100, activation="relu")(z)
codings_mean = keras.layers.Dense(10)(z)
codings_log_var = keras.layers.Dense(10)(z)
codings = Sampling()([codings_mean, codings_log_var]) # normal distribution
variational_encoder = keras.models.Model(inputs=[inputs], outputs=[codings])
```



## Variational Autoencoders (5/6)

### ► Decoder part

```
decoder_inputs = keras.layers.Input(shape=[codings_size])
x = keras.layers.Dense(100, activation="relu")(decoder_inputs)
x = keras.layers.Dense(150, activation="relu")(x)
x = keras.layers.Dense(28 * 28, activation="sigmoid")(x)
outputs = keras.layers.Reshape([28, 28])(x)
variational_decoder = keras.models.Model(inputs=[decoder_inputs], outputs=[outputs])
```



# Variational Autoencoders (6/6)

```
codings = variational_encoder(inputs)
reconstructions = variational_decoder(codings)
model = keras.models.Model(inputs=[inputs], outputs=[reconstructions])

latent_loss = -0.5 * K.sum(1 + codings_log_var - K.exp(codings_log_var)
                           - K.square(codings_mean), axis=-1)
model.add_loss(K.mean(latent_loss) / 784.)
```

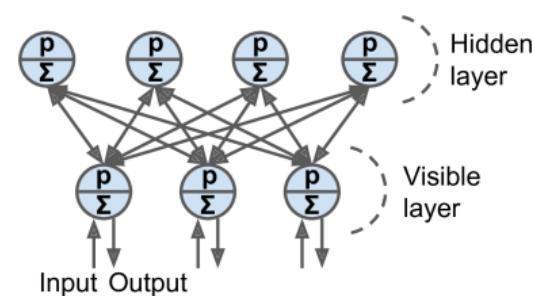




# Restricted Boltzmann Machines

# Restricted Boltzmann Machines

- ▶ A Restricted Boltzmann Machine (RBM) is a stochastic neural network.
- ▶ Stochastic meaning these activations have a probabilistic element, instead of deterministic functions, e.g., logistic or ReLU.
- ▶ The neurons form a bipartite graph:
  - One visible layer and one hidden layer.
  - A symmetric connection between the two layers.
  - There are no connections between neurons within a layer.





# Let's Start With An Example

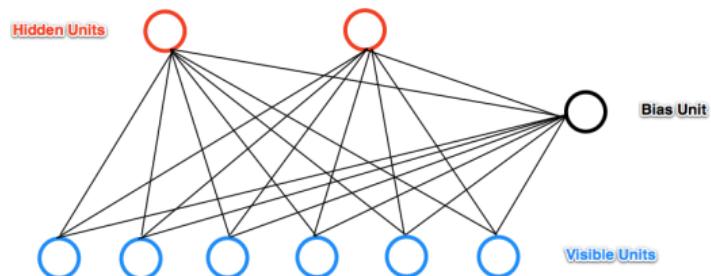
## RBM Example (1/10)

- ▶ We have a set of **six movies**, and we ask users to tell us which ones **they want to watch**.
- ▶ We want to learn two **latent neurons (hidden neurons)** underlying movie preferences, e.g., **SF/fantasy** and **Oscar winners**



## RBM Example (2/10)

- ▶ Our RBM would look like the following.



## RBM Example (3/10)

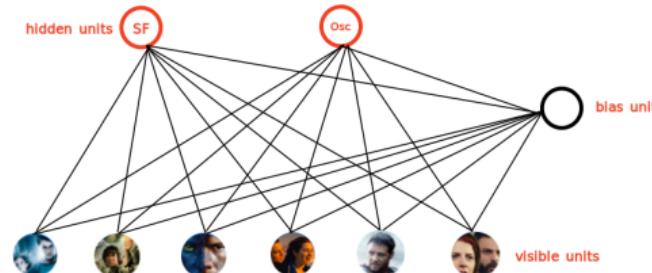
- ▶ Alice: (HP=1, Avatar=1, LOTR=1, Glad=0, Titan=0, Sep=0), Big SF fan.
- ▶ Bob: (HP=1, Avatar=0, LOTR=1, Glad=0, Titan=0, Sep=0), SF fan, but not Avatar.
- ▶ Carol: (HP=1, Avat=1, LOTR=1, Glad=0, Titan=0, Sep=0), Big SF fan.
- ▶ David: (HP=0, Avat= 0, LOTR=1, Glad=1, Titan=1, Sep=1), Big Oscar winners fan.
- ▶ Eric: (HP=0, Avat=0, LOTR=1, Glad=1, Titan=0, Sep=1), Oscar winners fan, but not Titanic.
- ▶ Fred: (HP=0, Avat=0, LOTR=1, Glad=1, Titan=1, Sep=1), Big Oscar winners fan.



## RBM Example (3/10)

- ▶ Assume the given input  $x_i$  is the 0 or 1 for each visible neuron  $v_i$ .
  - 1: like a movie, and 0: dislike a movie
- ▶ Compute the activation energy at hidden neuron  $h_j$ :

$$a(h_j) = \sum_i w_{ij} v_i$$



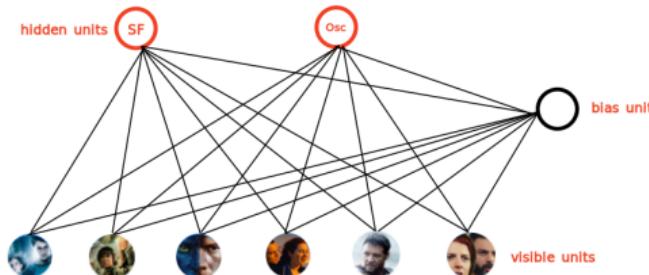
## RBM Example (4/10)

- ▶ For each hidden neuron  $h_j$ , we compute the probability  $p(h_j)$ .

$$a(h_j) = \sum_i w_{ij} v_i$$

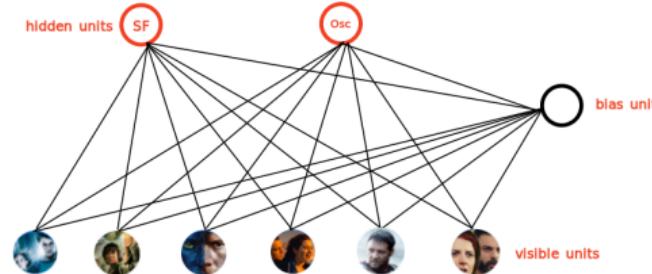
$$p(h_j) = \text{sigmoid}(a(h_j)) = \frac{1}{1 + e^{-a(h_j)}}$$

- ▶ We turn on the hidden neuron  $h_j$  with the probability  $p(h_j)$ , and turn it off with probability  $1 - p(h_j)$ .



## RBM Example (5/10)

- ▶ Declaring that you like Harry Potter, Avatar, and LOTR, doesn't guarantee that the SF/fantasy hidden neuron will turn on.
- ▶ But it will turn on with a high probability.
  - In reality, if you want to watch all three of those movies makes us highly suspect you like SF/fantasy in general.
  - But there's a small chance you like them for other reasons.





## RBM Example (6/10)

- ▶ Conversely, if we know that one person **likes SF/fantasy** (so that the SF/fantasy **neuron is on**)
- ▶ We can ask the RBM to generate a set of **movie recommendations**.
- ▶ The **hidden neurons** send messages to the **visible (movie) neurons**, telling them to **update their states**.

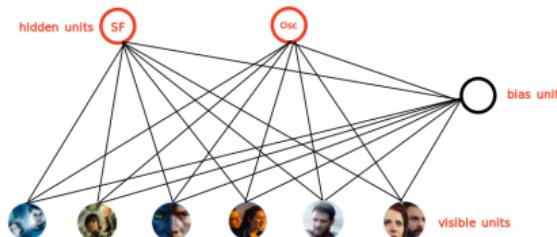
$$a(v_i) = \sum_j w_{ij} h_j$$

$$p(v_i) = \text{sigmoid}(a(v_i)) = \frac{1}{1 + e^{-a(v_i)}}$$

- ▶ Being on the **SF/fantasy** neuron **doesn't guarantee** that we'll always recommend all three of **Harry Potter**, **Avatar**, and **LOTR**.
  - For example **not everyone** who likes science fiction liked Avatar.

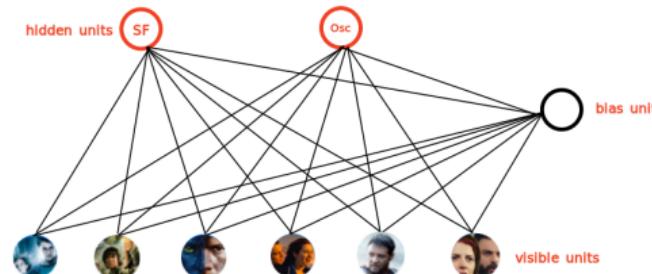
## RBM Example (7/10)

- ▶ How do we **learn** the **connection weights**  $w_{ij}$  in our network?
- ▶ Assume, as an input we have a bunch of **binary vectors  $x$**  with **six elements** corresponding to a **user's movie preferences**.
- ▶ We do the **following steps** in each **epoch**:
- ▶ 1. Take a **training instance  $x$**  and set the **states** of the **visible neurons** to these preferences.



## RBM Example (8/10)

- ▶ 2. Update the **states** of the **hidden neurons**.
  - Compute  $a(h_j) = \sum_i w_{ij} v_i$  for each **hidden neuron**  $h_j$ .
  - Set  $h_j$  to 1 with probability  $p(h_j) = \text{sigmoid}(a(h_j)) = \frac{1}{1+e^{-a(h_j)}}$
- ▶ 3. For each edge  $e_{ij}$ , compute **positive**( $e_{ij}$ ) =  $v_i \times h_j$ 
  - I.e., for each **pair of neurons**, measure whether they are **both on**.



## RBM Example (9/10)

- ▶ 4. Update the **state** of the **visible neurons** in a similar manner.
  - We denote the updated visible neurons with  $v'_i$ .
  - Compute  $a(v'_i) = \sum_j w_{ij} h_j$  for each **visible neuron**  $v'_i$ .
  - Set  $v'_i$  to 1 with probability  $p(v'_i) = \text{sigmoid}(a(v'_i)) = \frac{1}{1+e^{-a(v'_i)}}$
- ▶ 5. Update the **hidden neurons** again similar to step 2. We denote the **updated hidden neurons** with  $h'_j$ .
- ▶ 6. For each edge  $e_{ij}$ , compute **negative**( $e_{ij}$ ) =  $v'_i \times h'_j$



## RBM Example (10/10)

- ▶ 7. Update the weight of each edge  $e_{ij}$ .

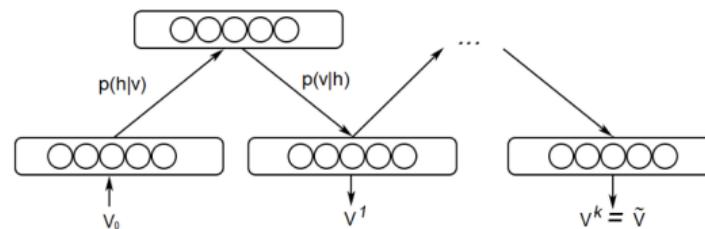
$$w_{ij} = w_{ij} + \eta(\text{positive}(e_{ij}) - \text{negative}(e_{ij}))$$

- ▶ 8. Repeat over all training examples.
- ▶ 9. Continue until the error between the training examples and their reconstructions falls below some threshold or we reach some maximum number of epochs.



## RBM Training (1/2)

- ▶ Step 1, Gibbs sampling: what we have done in steps 1-6.
- ▶ Given an input vector  $v$ , compute  $p(h|v)$ .
- ▶ Knowing the hidden values  $h$ , we use  $p(v|h)$  for prediction of new input values  $v$ .
- ▶ This process is repeated  $k$  times.





## RBM Training (2/2)

- ▶ Step 2, **contrastive divergence**: what we have done in step 7.
  - Just a fancy name for **approximate gradient descent**.

$$\mathbf{w} = \mathbf{w} + \eta(\text{positive}(\mathbf{e}) - \text{negative}(\mathbf{e}))$$



# More Details about RBM

# Energy-based Model (1/3)

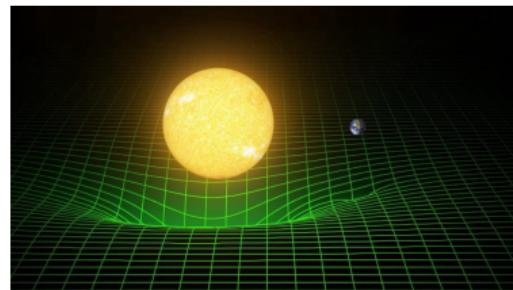
► Energy a quantitative property of physics.

- E.g., gravitational energy describes the potential energy a body with mass has in relation to another massive object due to gravity.



## Energy-based Model (2/3)

- ▶ One purpose of deep learning models is to **encode dependencies between variables**.
- ▶ The capturing of **dependencies** happen through associating of a **scalar energy** to each **state of the variables**.
  - Serves as a **measure of compatibility**.
- ▶ A **high energy** means a **bad compatibility**.
- ▶ An **energy based model** tries always to **minimize** a predefined energy function.

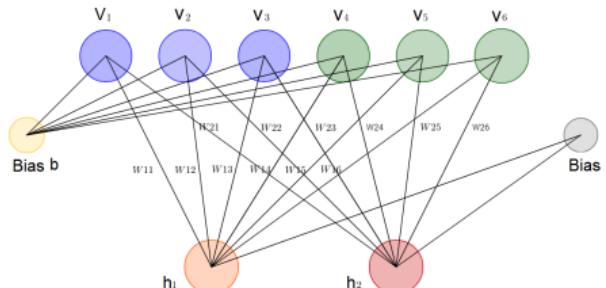


## Energy-based Model (3/3)

- ▶ The **energy function** for the RBMs is defined as:

$$E(\mathbf{v}, \mathbf{h}) = -\left(\sum_{ij} w_{ij} v_i h_j + \sum_i b_i v_i + \sum_j c_j h_j\right)$$

- ▶ **v** and **h** represent the **visible** and **hidden** units, respectively.
- ▶ **w** represents the **weights** connecting visible and hidden units.
- ▶ **b** and **c** are the **biases** of the visible and hidden layers, respectively.





## RBM is a Probabilistic Model (1/2)

- ▶ The probability of a certain state of  $v$  and  $h$ :

$$p(v, h) = \frac{e^{-E(v, h)}}{\sum_{v, h} e^{-E(v, h)}}$$

- ▶ In physics, the joint distribution  $p(v, h)$  is known as the **Boltzmann Distribution** or **Gibbs Distribution**.
- ▶ At each point in time the RBM is in a certain state.
  - The state refers to the values of neurons in the visible and hidden layers  $v$  and  $h$ .

## RBM is a Probabilistic Model (2/2)

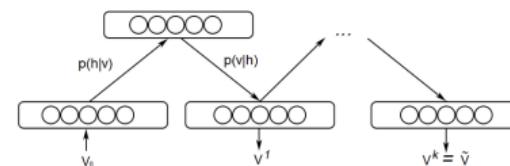
- ▶ It is **difficult** to calculate the **joint probability** due to the **huge number of possible combination** of **v** and **h**.

$$p(v, h) = \frac{e^{-E(v,h)}}{\sum_{v,h} e^{-E(v,h)}}$$

- ▶ Much easier is the calculation of the **conditional probabilities** of state **h** given the state **v** and vice versa (**Gibbs sampling**)

$$p(h|v) = \prod_i p(h_i|v)$$

$$p(v|h) = \prod_j p(v_j|h)$$





## Learning in Boltzmann Machines (1/2)

- ▶ RBMs try to learn a probability distribution from the data they are given.
- ▶ Given a training set of state vectors  $\mathbf{v}$ , learning consists of finding parameters  $\mathbf{w}$  of  $p(\mathbf{v}, \mathbf{h})$ , in a way that the training vectors have high probability  $p(\mathbf{v})$ .

$$p(\mathbf{v}|\mathbf{h}) = \frac{\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}}{\sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}}$$

- ▶ Use the maximum-likelihood estimation.
- ▶ For a model of the form  $p(\mathbf{v})$  with parameters  $\mathbf{w}$ , the log-likelihood given a single training example  $\mathbf{v}$  is:

$$\log p(\mathbf{v}|\mathbf{h}) = \log \frac{\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}}{\sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}} = \log \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} - \log \sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}$$

## Learning in Boltzmann Machines (2/2)

- ▶ The **log-likelihood gradients** for an RBM with **binary units**:

$$\frac{\partial \log p(\mathbf{v}|\mathbf{h})}{\partial w_{ij}} = \text{positive}(e_{ij}) - \text{negative}(e_{ij})$$

- ▶ Then, we can **update** the weight **w** as follows:

$$w_{ij}^{(\text{next})} = w_{ij} + \eta(\text{positive}(e_{ij}) - \text{negative}(e_{ij}))$$



**IT'S OVER**

**IT'S FINALLY OVER**

[makeameme.org](http://makeameme.org)



# Summary



# Summary

- ▶ Autoencoders
  - Stacked autoencoders
  - Denoising autoencoders
  - Variational autoencoders
- ▶ Restricted Boltzmann Machine
  - Gibbs sampling
  - Contrastive divergence



## Reference

- ▶ Ian Goodfellow et al., Deep Learning (Ch. 14, 20)
- ▶ Aurélien Géron, Hands-On Machine Learning (Ch. 17)



# Questions?